The Effect of AOP on Software Engineering, with Particular Attention to OIF and Event Quantification

Robert E. Filman
RIACS
NASA Ames Research Center
Moffett Field, CA 94035
rfilman@arc.nasa.gov

Klaus Havelund
Kestrel Technologies
NASA Ames Research Center
Moffett Field, CA 94035
havelund@email.arc.nasa.gov

January 31, 2003

Abstract

We consider the impact of Aspect-Oriented Programming on Software Engineering, and, in particular, analyze two AOP systems, one of which does component wrapping and the other, quantification over events, for their software engineering effects.

1 Introduction

Developing software is difficult. It requires dealing an inherently complex and diverse problem space, amidst the sands of wavering requirements and a shifting environments. The fundamental plasticity of software encourages its engineering abuse. (No one asks the designer of an almost completed house to add another story and relocate the kitchen, but software projects often face equivalent demands.) Additionally, most software must be engineered to fit into an environment with intensive and extensive human interaction. Engineering with so fallible subsystems as humans compounds the difficult of building correct and reliable systems.

We have written at length on the nature of Aspect Oriented Programming (AOP) [1, 3]. Aspect Orientation has evolved as a technology for combining separately created software components into working systems. In contrast to conventional mechanisms for such combination (e.g., subprograms, inheritance) AOP allows "surgical" mechanisms, where the behaviors of the separately specified systems intertwine without explicit invocation, and where a single intertwined behavior can be applied "wherever needed" in a developing system. In this paper, we consider the impact of AOP on software engineering, examining AOP in general and two particular AOP systems that we have worked on: the Object Infrastructure Framework [2], a completed project, and our current not-yet-named work on realizing AOP through quantification over events [4].

What do we want from our software systems? A host of *ilities*, including correctness, efficiency, maintainability, portability, reliability, interoperability, fault tolerance, recoverability, learnability, analyzability, adaptability, reusability, robustness, testability, verifyability, comprehensibility, consistency, traceability, evolvability, measurability, and modularity. Aspect Orientation can help with some of these ilities, while being an impediment to others. Software development is not a monolithic activity. Instead, it is better understood as a collection of not-necessarily sequential activities:

Concept exploration An abstract determinination of what it is one is trying to build and whether it is possible to develop said system.

Requirements analysis A more specific clarification of the desired behavior, including the development, operating and maintenance environments, the conceptual model of what is being acted upon, the shape of the user interface, the functional and non-functional behavior of the system, how it handles errors, and what can be foreseen about its evolution.

Specification A more detailed description of the proposed modules and their interfaces.

Design A description of how to build the proposed system.

Implementation The actual system construction, including not only coding but also configuration management and unit debugging.

Testing Actions to create the misapprehension that the system actually works as desired, occasionally punctured by moments of realization that what was specified is not really what is desired.

Maintenance Fixing the system to make its behavior closer to what it is supposed to be.

Evolution Changing the system to match changes in the requirements and execution environment.

As people are notably poor at understanding the consequences of their assumptions, and even such local omniscience doesn't control an independently varying external environment, pragmatically, these steps crosscut in interesting ways.

2 AOP and software engineering

Where can AOP (and AOSD) effect these steps? AOP offers its greatest promises (and deepest pitfalls) in the implementation and evolution elements of system development. We consider, for "generic" AOP, the effect of AOP on a variety of classical programming development technologies. These technologies have been argued as easing software creation and evolution.

Modularization Modularization is breaking elements into separate pieces, where there is considerable *cohesion* (strength of relationships between elements, include data-type, functional, logical and sequential cohesion) within the module and minimal *coupling* (linkage between modules based on the communication between them, such as data definition, data element, control, content and global coupling). Modularization is a gold-standard of programming language mechanisms to aid software engineering. Examples of Java modularization mechanisms are packages and objects. Examples of Java coupling mechanisms are inheritance and method invocations. Generic AOP gets a mixed score on modularization—it offers the promise of removing to their own modules separate concerns, and the problem that the coupling between modules (as measured by their interaction) may increase exponentially. That is, rather than interacting at well-defined method invocations, AOP systems may have interactions through the system.

Data abstraction A hallmark of modern thought is that of data abstraction—keeping internal structures hidden from the users of those structures. This also include explicit visibility controls, such as Java's `private` and `protected` and C++'s `friend`. Like modularization, generic AOP is a mixed bag on data abstraction—it allows abstracting out even finer grained activities, while (in some systems) providing access to internal structures that conventional gospel would argue should remain hidden.

Genericity A holy grail of software engineering is reuse. We'd like our languages to supply mechanisms for reusing the same code in different contexts. Conventional mechanisms for reuse include generics, modules, and parameters. Genericity is also a holy grail for AOP; some kinds of surgery find themselves being quite specific about the application to which they apply. Quantification is essential for reusable aspects.

Program semantics A mark of good design is that individual program elements have a well-defined and specific semantics. AOP-based program surgery undermines this concept.

Debugability Conventional languages keep a close correlation between the source code and what executes. That debugging optimized code is a difficult task [5] is evidence of the importance of this correspondence. AOP, with its threat to scatter the cause of execution throughout the runtime process is a similar (though perhaps not a complex) threat. On the other hand, the AOP ability to easily instrument code proves a boon to debugging.

Static programming certifications A topic of considerable energy in programming language development is the degree to which classes of errors can be recognized at compile time by the annotation of a program with additional notation (static analysis). The most recognizable of such mechanisms are static type checking and lint. However, in some languages, certain other assertions and pragmas can have a similar effect. This desire to perform static type checking works against the natural programming desire to write more expressive and encompassing systems. Relatively simple AOP languages defeat static type checking; more elaborate ones put considerable energy into preserving type mechanisms. AOP itself could be the basis for generically implementing dynamic assertion checking.

Software metrics A subfield of software engineering is concerned with measuring the textual properties of code to get insight into its complexity, development cost and maintainability. AOP will likely require new metrics and new tools for computing those metrics.

Traceability An important element of general software engineering is the ability to trace requirements to the actual code. By separating concerns, AOP should aid traceability.

Software development tools A large variety of software development tools for activities such as graphical design, editing, browsing, pretty printing and searching have been developed. Generic AOP, with its perversion of the original code intent, demands modification of such tools.

Testing Testing typically encompasses unit testing, integration testing, system testing and regression testing. As free-float, non-executable code, most generic aspects resist unit testing. Aspects should not have much effect on the other elements.

Configuration management Configuration management include build, change, and version control. Aspects should slightly complicate this issue, as they suggest more steps to building, more things to keep track of and more alternative versions to manage. However, that's what such tools are supposed to be good for.

3 Object Infrastructure Framework

The Object Infrastructure Framework (OIF) [2] was an AOP system that worked by wrapping components with a structure that provided pluggable behaviors in the wrappers. OIF realized the following key ideas:

Intercepting communications OIF intercepts and manipulates communications among functional components, invoking appropriate "services" on these calls. Semantically, this is equivalent to wrapping or filtering on both the client and server side of a distributed system. The next five points can be understood as describing the architecture of a flexible wrapping system.

Discrete injectors Our communication interceptors are first class objects: discrete components that have (object) identity and are invoked in a specific sequence. We call them injectors. In a distributed system, an ility may require injecting behavior on both the client and the server. Injectors are uniform so we can build utilities to manipulate them.

Injection by object/method Each instance and each method on that object can have a distinct sequence of injectors.

Dynamic injection The injectors on an object/method are maintained dynamically and can, with the appropriate privileges, be added and removed. Examples of the uses of dynamic configuration include placing debugging and monitoring probes on running applications and creating software that detects its own obsolescence and updates itself.

Annotations Injectors can communicate among themselves by adding annotations to the underlying requests of the procedure call mechanism.

Thread contexts One OIF goal was to keep the injection mechanism invisible to the functional components (or at least to those functional components that want to remain ignorant of it.) To allow clients and servers to communicate with the injector mechanism, the system maintains a "thread context" of annotations for threads, and copies between this context and the annotation context of requests. Thread contexts and annotations together provide the data space for communication between the application and injectors and among injectors. (Injectors generated by the same factory or industrial complex can also share a data space defined by their factory structure.)

High-level specification compiler To bridge the conceptual distance between abstract ilities and discrete sequences of injectors, we created a compiler from high-level specification of desired properties and ways to achieve these properties to default injector initializations.

OIF had the following characteristics with respect to software engineering

Modularization OIF injectors were clear modules, and could be (and were) reusable. There were mechanisms for forming collection of related injectors (for example, encrypt/decrypt pairs), but these mechanisms relied on the structuring mechanisms of the underlying programming language.

Data abstraction OIF injectors worked around existing components, and thus didn't violate their privacy.

Program semantics As wrappers, OIF injectors could modify the semantics of a component, but at least this modification happened in a single place.

Debugability Injectors executed in a well-defined place in the execution sequence. This kept them from being much of a burden on the debugging process. In fact, specific injectors were developed specifically as aids to debugging, and the ability to dynamically add injectors for individual objects is a powerful debugging mechanism.

Static programming certifications This was perhaps the weakest software-engineering element of OIF. Its mechanism relied on programmers understanding and properly using the types of their systems.

Software metrics As a minimally intrusive mechanism, and one which was primarily realized in code, OIF should have had minimal impact on metric mechanisms.

Traceability This was a prime theme of OIF.

Software development tools As a minimally intrusive mechanism, OIF did not have much impact on software development tools.

Testing OIF suffered from the need to create unit testing elements for injectors.

Configuration management OIF posed no special static configuration management issues. However, the ability to dynamically insert and remove injectors from particular components suggested both novel configuration management headaches and perhaps the prospect of systems that could automatically adapt their configuration through system evolution.

4 Quantification over Events

Quantification over events (QE) [4] is a nascent system that proposes to do AOP by

1. Providing a language for describing sequences of events of interest in the execution of a program and actions to be taken on these events.
2. Providing a transformation mechanism that takes a program and produces a transformed program such that when the specified events occur, the transformed code will also execute the associated actions.

This is a powerful notion, which, properly applied, is capable of wrecking havoc on almost all software engineering principles.

Modularization In QE related transforms can be modularized, and, like the generic AOP claim, nicely separated.

Data abstraction QE provides the opportunity to completely strip the data abstraction from a given module.

Program semantics As a program transformation tool, QE has the ability to pervert the semantics of any given module to its heart's content.

Debugability QE transforms themselves are likely to prove to be difficult to debug. However, as a debugging tool, QE offers the possibility of tremendous insight into program execution.

Static programming certifications Making transformations that conform to the static programming certifications of the underlying language will be a major source of complexity in writing QE transformations.

Software metrics Determining the right combination of metrics for transformations and base programs is a subject for further research.

Traceability QE has the potential for spanning the entire traceability range. Behaviors may be traced to the particular transformation that caused them, which is good. However, arbitrary use of transformation may destroy the traceability of other parts of the code.

Software development tools Since QE produces source code, many existing software analysis tools will still be able to work on that code. To the extent that such tools want human comprehensible code, QE output may be difficult to understand.

Testing Like OIF, QE will suffer from the need to create unit testing elements for transformations. On the other hand, QE can be used to introduce a testing regime.

Configuration management QE will pose the normal generic AOP configuration management issues.

5 Closing Remarks

We have identified those parts of the software engineering process most affected by aspect technology, and examined two aspect systems with respect to these issues. As illustrated by the two systems, aspects are a two-sides weapon, capable of introducing beneficial abstractions and separations, but also liable to destroying structures demanded by other parts of the software engineering process. The particular software-engineering effects of QE suggest that it may parallel Knuth's goals for TeX—as an environment in which to build other AOP systems, rather than an unsophisticated end-user system in itself, suggesting that some day, the appropriate AOP language may be a LaQE.

References

[1] R. Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, June 2001.

[2] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden. Inserting ilities by controlling communications. *Comm. ACM*, 45(1):116–122, Jan. 2002.

[3] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[4] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002)*, pages 45–49, Mar. 2002.

[5] C. Tice and S. L. Graham. Optview: a new approach for examining optimized code. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 19–26. ACM Press, 1998.